IMT Mines Albi-Carmaux
École Mines-Télécom

Projet Système d'Information

Rendu phase 3

Rapport explicatif du code source de Tutor Link

Groupe OEM

Muneeb MIRZA - Julien LAGARDE - Ghali SLAOUI - Meanonn TAN

Sommaire

Environnement, Configuration et Arborescence

Environnement

Notre projet en système d'information consiste à produire une application web qui répond au cahier des charges posté lors de la phase-1. Les outils techniques de ce projet nous ont été fixés. Nous devons donc utiliser les outils suivants :

- Le framework Flask de Python est utilisé pour le développement de l'application
- Nous avons une base de données sur l'application PostgreSQL
- Nous utilisons SQLalchemy pour interagir avec cette base de données via notre code source Python.

Nous développerons plus en détails les autres librairies nécessaires au cours du projet ultérieurement.

Configuration

L'application utilise deux variables d'environnement: `FLASK_APP` et `TUTORLINK_SETTINGS`.

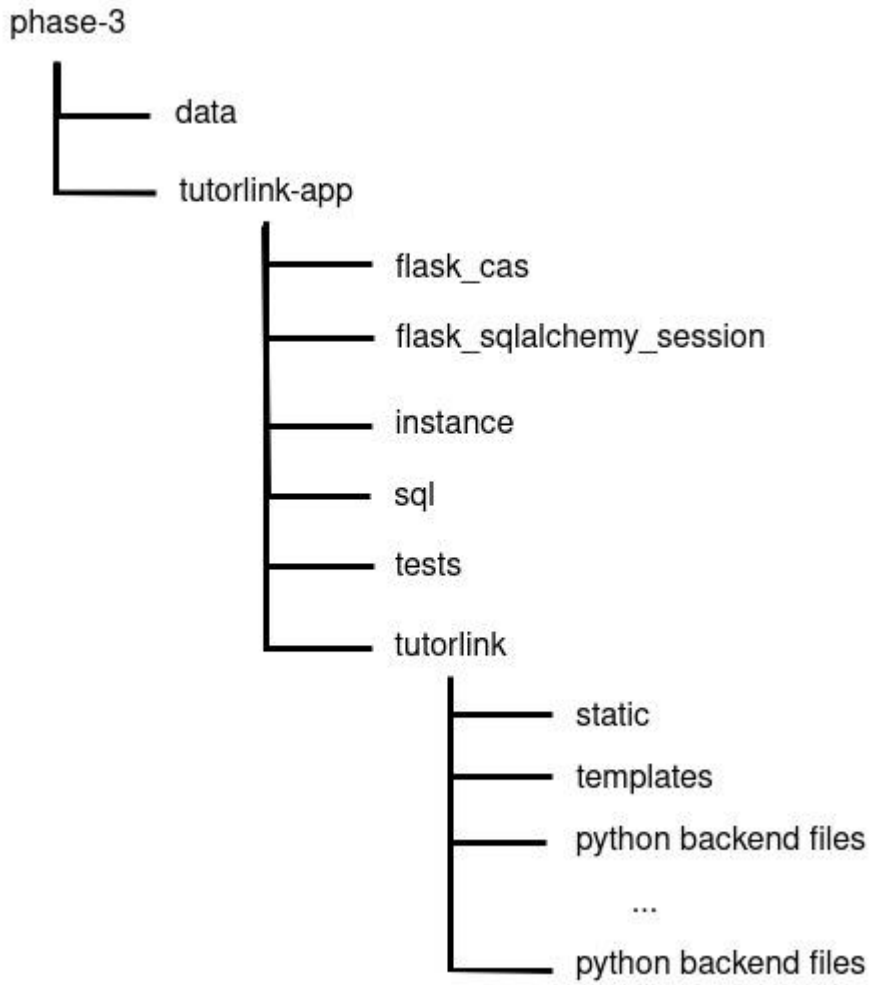
La 1re variable d'environnement (`FLASK_APP`) est définie dans le fichier `.flaskenv`. Sa valeur est utilisée par flask pour nommer l'application (ici, le nom de l'application est `tutorlink`).

La 2e variable d'environnement (`TUTORLINK_SETTINGS`) est définie à chaque lancement via la ligne de commande. Elle permet de définir le nom du fichier de configuration à aller chercher dans le répertoire `instance/`. Dans ce répertoire il n'y a que les fichiers `development.py` et `test.py`. Seuls les objets dont le nom est en majuscules sont récupérés depuis le fichier de configuration et sont ensuite utilisables par l'application (via `app.config`).

Un fichier de configuration `setup.cfg` est présent à la racine du projet contenant les metadata du projet, les modules python requis et donc à installer dans l'environnement virtuel, ou encore pour pouvoir effectuer les tests de l'application. Ce fichier est pris en compte par l'application grâce au fichier `setup.py`.

Arborescence

Voici l'arborescence du code source de notre application. Nous supposerons que le dossier "phase-3" correspond à la racine de notre projet.



Root :

- **data :** Dossier contenant les exports json qui ont été faits depuis Synapses.
- **tutorlink-app :** Dossier avec le code source de l'application
    - **flask_cas :** Dossier contenant tout le code relatif à l'utilisation de l'authentification avec CAS.
    - **flask_sqlalchemy_session :** Dossier contenant un script python qui fournit une session SQLAlchemy à portée d'application qui crée des sessions uniques par requête flask.
    - **instance :** Dossier contenant les fichiers python de configuration avec les variables d'environnement et l'URI de la base de données qui sera utilisée pour l'application et les tests.
    - **sql :** Dossier contenant les scripts sql pour le schéma de la base de données et pour peupler celle-ci.
    - **tests :** Dossier contenant les scripts python de tests.

- **tutorlink :** Dossier avec les codes front-end et back-end de l'application.
  - **static :** Dossier qui contient les fichiers CSS et javascript pour le front-end de l'application.
  - **templates :** Dossier avec tous les fichiers html.
  - **python files :** Ensemble des fichiers python concernant le back-end de l'application (principalement blueprints et décorateurs).

Explication détaillée des dossiers autour du code source

Base de données - /sql

Nous trouvons dans le dossier `/sql` les fichiers `schema.sql` ainsi que `populate.sql`.

`schema.sql` :

Ce code contient les instructions pour initialiser le schéma de la base de données.

Les instructions commencent par supprimer les tables existantes pour éviter les erreurs. Ensuite, elles créent les tables et définissent les relations entre elles à l'aide de clés.

Les tables créées sont les suivantes :

- role : elle stocke les différents rôles que peut avoir un utilisateur (enseignant, étudiant, administrateur, etc.).
- session_type : elle stocke les différents types de sessions (cours magistral, TD, TP, etc.).
- user : elle stocke les informations sur les utilisateurs (nom, prénom, adresse email, rôle, etc.).
- module : elle stocke les informations sur les différents modules (nom, label, description, etc.).
- ue : elle stocke les informations sur les différentes UE (unités d'enseignement) de l'université (nom, label, etc.).
- session : elle stocke les informations sur les différentes sessions (date de début, date de fin, type, module associé, UE associée, etc.).

- favorite : elle stocke les informations sur les modules préférés des utilisateurs.
- lectured_by : elle stocke les informations sur les enseignants qui donnent font une session.
- managed_by : elle stocke les informations sur les enseignants qui gèrent un module.

`populate.sql` :

Ce script permet de remplir les tables citées précédemment dans `schema.sql` avec, pendant la phase de développement et de tests, des données tests, mais également à remplir certains utilisateurs en tant qu'admin dans la base de données. En effet, seul un administrateur peut donner les droits d'administrateur à un autre utilisateur (cette fonctionnalité sera expliquée plus tard). Du coup pour le premier administrateur de l'application, il faut utiliser ce script de remplissage de la base de données.

Interaction avec la base de données - /flask_sqlalchemy_session

On trouve dans ce dossier un fichier `__init__.py` :

Dans ce code, une classe appelée `flask_scoped_session` est définie en tant que sous-classe de `scoped_session` de SQLAlchemy.

La méthode `init_app` de la classe flask_scoped_session configure la création et la suppression d'une session SQLAlchemy pour notre application passée en paramètre. Lorsque l'application est initialisée, une instance de la classe `flask_scoped_session` est créée et attachée à l'application en tant qu'attribut `scoped_session`.

La fonction `_get_session` utilise la pile de contexte d'application Flask (`_app_ctx_stack`) pour récupérer la session SQLAlchemy en cours. Cette fonction est utilisée pour créer un objet `LocalProxy` appelé `current_session`, qui permet d'accéder à la session SQLAlchemy en cours dans le contexte d'une requête de notre application Flask.

Authentification via CAS - /flask_cas

Ce dossier permet d'utiliser l'authentification CAS qui permet d'accéder à l'application Tutorlink. Dans ce dossier nous retrouvons les scripts `cas_urls.py`, `routing.py` et un `__init__.py`.

`cas_urls.py` :

Ce script permet de définir des fonctions pour créer des URL pour accéder au service d'authentification Central Authentication Service (CAS). Il définit trois fonctions principales :

- `create_cas_login_url` : Crée une URL de connexion CAS.
- `create_cas_logout_url` : Crée une URL de déconnexion CAS.
- `create_cas_validate_url` : Crée une URL de validation CAS.

Chacune de ces fonctions prend en entrée les informations nécessaires pour construire une URL CAS, telles que l'URL de base du service CAS, le chemin d'accès à l'application utilisant CAS, les informations de service (URL de retour), et des paramètres facultatifs comme "renew" et "gateway". Les fonctions utilisent ensuite la fonction `create_url` pour construire l'URL finale en combinant ces informations et en effectuant l'encodage nécessaire.

`routing.py` :

Ce script contient le code qui définit complètement le blueprint “cas”. On y trouve 3 routes qui sont configurées.

La fonction login est utilisée pour rediriger l'utilisateur vers le système CAS pour l'authentification. Si la connexion réussit, le système CAS renvoie un ticket qui est ensuite validé à l'aide de la fonction validate.

La fonction validate tente de valider le jeton et renvoie `True` si la validation réussit, `False` sinon. Les résultats de la validation sont stockés dans les clés `CAS_USERNAME_SESSION_KEY` et `CAS_ATTRIBUTES_SESSION_KEY` de la session Flask. La fonction utilise la bibliothèque `xmltodict` pour analyser les résultats de la validation, qui sont renvoyés au format XML. La bibliothèque `xmltodict` est ajoutée pour cette raison dans le fichier de config `setup.cfg` à la racine du projet.

La fonction `logout` est utilisée pour déconnecter l'utilisateur de l'application Flask.

Tests réalisés sur notre application - /tests

On trouve dans ce dossier tous les tests réalisés pour le développement de l'application :

`conftest.py` :

Ce code fait office de fichier de configuration pour les tests utilisant pytest. Il initialise des variables d'environnement, des bases de données, des fichiers JSON et des objets SQLAlchemy pour les tests unitaires.

Plus précisément, ce fichier de configuration contient des fonctions de fixation (fixtures) qui créent des instances réutilisables de l'application de test, du client web pour simuler des requêtes provenant d'un navigateur, des objets pour l'accès à la base de données et des fichiers JSON pour la population de la base de données de test.

Le code utilise également des commandes pour exécuter des scripts SQL à l'aide de l'outil psql (PostgreSQL interactive terminal) afin d'initialiser les structures de la base de données de test avec les données requises.

En somme, le fichier de configuration permet d'initialiser les ressources nécessaires aux tests unitaires.

`test_admin_panel.py` :

Ce script contient les tests unitaires pour les routes "`admin_panel`", "`grant_admin`", et "`revoke_admin`". Il est destiné à tester les autorisations d'accès à ces routes, en particulier pour les utilisateurs disposant de différents rôles. Les tests s'assurent également que le rôle "`admin`" est correctement attribué et retiré aux utilisateurs.

La plupart des tests commencent par une étape de configuration dans laquelle la session d'un utilisateur est simulée, en lui attribuant un nom d'utilisateur fictif ("`CAS_USERNAME`"). Ensuite, une requête GET ou POST est effectuée pour accéder à une route spécifique ("`admin/panel`", "`admin/grant_admin/{username}`", ou "`admin/revoke_admin/{username}`"), en suivant les redirections si nécessaire. Les assertions sont ensuite utilisées pour vérifier que la réponse HTTP a le code de statut attendu (200, 403, etc.), et que la réponse contient des messages flash appropriés. Les tests de "`grant_admin`" et "`revoke_admin`" vérifient également que le rôle "`admin`" est bien attribué ou retiré aux utilisateurs dans la base de données.

test_auth.py :

Ce code est un ensemble de tests pour la route de création d'utilisateur "/auth/register".

La fonction "register_new_user" est utilisée pour enregistrer un nouvel utilisateur en simulant la connexion CAS et en envoyant des données via une requête POST à la route "/auth/register". Cette fonction est appelée dans les tests suivants:

- test_01_register_new_user : test de la création d'un nouvel utilisateur avec des données valides.
- test_02_register_without_data : test de la création d'un nouvel utilisateur sans envoyer de données.
- test_03_register_without_name : test de la création d'un nouvel utilisateur sans le nom.
- test_04_register_with_inexistant_role : test de la création d'un nouvel utilisateur avec un rôle inexistant.
- test_05_register_with_existing_username : test de la création d'un nouvel utilisateur avec un nom d'utilisateur existant.
- test_06_register_redirect_user_missing_data : test de redirection de l'utilisateur lorsqu'il manque des données.

Dans chacun de ces tests, on vérifie le code de réponse de la requête, on vérifie les messages flash retournés par la page, on vérifie que l'utilisateur a bien été créé dans la base de données, et on effectue d'autres vérifications spécifiques à chaque test.

test_data_update.py :

- test_01_data_update_route : teste si la page de mise à jour des données peut être affichée pour un utilisateur connecté avec un rôle d'administrateur.
- test_02_data_update_route_with_bad_role : teste si la page de mise à jour des données peut être affichée pour un utilisateur connecté avec un rôle non administrateur.
- test_03_data_update_route_no_login : teste si la page de mise à jour des données peut être affichée pour un utilisateur non connecté.
- test_04_parse_users : teste la fonction parse_users qui permet de parser une chaîne de caractères représentant un utilisateur en trois parties : nom, prénom et identifiant.
- test_05_read_json : teste la fonction read_json qui permet de lire le fichier JSON contenant les données issues de Synapses.

- `test_06_data_update_no_file` : teste le cas où aucun fichier n'est sélectionné pour la mise à jour des données. Le test envoie une requête POST à l'URL `/data/update` avec des données vides et vérifie si la réponse est un code de statut HTTP 200. Il vérifie également si un message flash est affiché avec une catégorie "`warning`" et le message "`No file selected`".
- `test_07_data_update` : teste la mise à jour des données avec un fichier JSON valide.
- `test_08_data_update_add_lecturer` : teste la mise à jour des données avec l'ajout d'un intervenant à une session existante.
- `test_09_data_update_confirm_lecturer` : teste la mise à jour des données avec la confirmation d'un intervenant existant pour une session.
- `test_10_data_update_new_session` : teste l'ajout d'une nouvelle session à la base de données.

`test_db.py` :

- `create_user_in_db` : cette fonction crée un nouvel utilisateur dans la base de données. Si `no_data` est `True`, elle crée simplement un utilisateur avec un nom d'utilisateur username. Sinon, elle crée un utilisateur avec des informations supplémentaires telles que l'e-mail, le nom, le prénom, le rôle et le statut administrateur. La fonction utilise l'ORM de SQLAlchemy pour créer un objet `User` à partir des informations fournies, puis elle ajoute cet objet à la session de base de données et commit les changements.
- `delete_user_in_db` : cette fonction supprime un utilisateur de la base de données. Si l'argument user est fourni, elle supprime cet utilisateur en utilisant l'ORM de SQLAlchemy. Sinon, elle recherche l'utilisateur avec le nom d'utilisateur username en utilisant une requête SQL et le supprime s'il existe. La fonction utilise également la méthode `commit()` pour enregistrer les changements dans la base de données.
- `test_01_consultation` : cette fonction teste l'accès à la base de données en exécutant une requête SQL pour récupérer tous les utilisateurs de la base de données, puis en parcourant chaque utilisateur pour vérifier que ses modules préférés et gérés sont bien des objets de type Module. La fonction utilise l'ORM de SQLAlchemy pour exécuter la requête et récupérer les résultats.
- `test_02_create_and_delete_user` : cette fonction teste la création et la suppression d'un utilisateur dans la base de données. Elle utilise la fonction `create_user_in_db` pour créer un nouvel utilisateur, puis elle vérifie que l'utilisateur a été créé avec succès en utilisant une requête SQL pour récupérer l'utilisateur avec le nom d'utilisateur username. Elle supprime

l'utilisateur en utilisant la fonction `delete_user_in_db` et vérifie qu'il a bien été supprimé de la base de données en utilisant une requête SQL pour vérifier que l'utilisateur n'existe plus.

`test_home.py` :

- `test_01_consultation_no_login` vérifie que lorsqu'un utilisateur non connecté essaie d'accéder à la page d'accueil, il est redirigé vers la page de connexion. Le test vérifie également que les routes '`home.index`' et '`index`' renvoient toutes deux à `'/'`.
- `test_02_consultation` vérifie que la page d'accueil est accessible une fois qu'un utilisateur est connecté. Il simule la connexion de l'utilisateur en ajoutant un nom d'utilisateur à la session.
- `test_03_doctorant` vérifie l'accès à la page d'accueil pour un utilisateur connecté en tant que doctorant. Le test vérifie également que la page contient un élément avec la classe CSS "circular-progress" qui correspond au dashboard du suivi des cours des doctorants.
- `test_04_enseignant` vérifie l'accès à la page d'accueil pour un utilisateur connecté en tant qu'enseignant. Le test vérifie également que la page contient un élément contenant la chaîne de caractères `"/∞"`.

`test_profile.py` :

- `test_01_profile_edit` vérifie si un utilisateur peut modifier son profil avec des données valides.
- `test_02_profile_edit_without_data` vérifie si l'utilisateur reçoit une erreur lorsque des données manquent.

Code source de l’application

CSS et Javascript - /static

Ce dossier contient les fichiers statiques (fichier CSS et javascript) nécessaires pour l'apparence de certaines classes utilisées dans les templates HTML.

Front end - /templates

Le dossier /templates contient les fichiers HTML qui régissent l'apparence de chaque vue, reliées et configurées par un blueprint qui définit les routes. Dans ce dossier on trouve un sous-dossier par blueprint avec le nom du blueprint, la base que l'on retrouvera sur toutes les pages ainsi que quelques vues qui ne correspondent pas forcément à un blueprint.

`base.html :`

La balise head contient les informations importantes de la page, tels que le titre de la page, l'encodage des caractères, la vue de la page et les liens vers les fichiers CSS et JS nécessaires pour le design de la page.

Le contenu principal de la page se trouve dans la balise body. Il y a d'abord un appel à une inclusion `topbar_banner.html`, qui contient la bannière en haut de la page.

Ensuite, on trouve une div avec une classe container-fluid qui est utilisée pour contenir le reste du contenu de la page. Dans cette div, il y a une inclusion alerts.html qui permet d'afficher des messages d'alerte à l'utilisateur.

La partie principale de la page est placée entre les balises {% block content %} et {% endblock %}, qui permettent de définir un contenu spécifique pour chaque page héritant de ce modèle de base.

En fin de page, on trouve les liens vers les fichiers JS nécessaires pour le fonctionnement de la page. On trouve également une ligne de code qui initialise le plugin Bootstrap tooltip. Enfin, il y a les liens vers les fichiers CSS et JS nécessaires pour le plugin Bootstrap Select.

`topbar_banner.html :`

Ce script permet l'affichage d'une barre supérieure (topbar) contenant un menu déroulant permettant de voir le profil de l'utilisateur et de se déconnecter. Cette barre supérieure n'apparaît que si l'utilisateur est connecté. Le deuxième élément est une

bannière avec une barre de navigation, qui contient un logo et des liens vers différentes pages. Cette barre de navigation n'apparaît que si l'utilisateur est connecté. Si l'utilisateur a également un rôle administrateur, deux liens supplémentaires apparaissent pour la mise à jour des données et l'accès au panneau d'administration.

`alerts.html :`

Ce code est un ensemble de macros utilisées pour afficher des messages d'alerte dans une application web Flask.

La première macro, "`alert`", prend plusieurs paramètres tels que le message à afficher, la catégorie de l'alerte (par défaut "`primary`"), une icône (facultative) et sa couleur (également facultative). Cette macro génère une balise HTML div contenant l'alerte avec le message, la catégorie, l'icône et la couleur spécifiées.

La deuxième partie du code utilise la macro "`alert`" pour afficher les messages d'alerte stockés dans la variable "`messages`" (qui est remplie par les messages flash dans une application Flask). Les alertes sont ensuite affichées dans une grille.

Le reste des scripts sont les vues correspondant aux différents blueprints qui sont détaillés dans la partie suivante.

Back end - fichiers python dans /tutorlink

Nous allons maintenant expliquer le fonctionnement de nos différents blueprints.

`__init__.py :`

Ce code configure l'application avec les paramètres de connexion au service d'authentification CAS et charge également des paramètres de configuration depuis un fichier d'environnement `'TUTORLINK_SETTINGS'`.
L'application crée plusieurs routes et associe chaque route à un blueprint, qui contient les fonctions pour gérer les requêtes entrantes. Les blueprints sont enregistrés sur l'application Flask.

La fonction `init_app()` initialise une base de données pour stocker des informations relatives à l'application.

Enfin, la fonction `create_app()` retourne l'application Flask créée.

db.py :

Ce code définit une fonction `connect_db` qui crée une connexion à une base de données en utilisant SQLAlchemy et Flask. L'automapping est utilisé pour mapper les tables de la base de données aux classes Python. Le code définit également les relations entre les classes mappées en utilisant des objets SQLAlchemy tels que relationship et back_populates.

Plus précisément, le code importe plusieurs modules SQLAlchemy et Flask, y compris `create_engine`, `MetaData`, `automap_base`, `relationship`, et `sessionmaker`. Il importe également des fonctions spécifiques pour l'automapping, telles que name_for_collection_relationship et name_for_scalar_relationship.

Le code définit une série de classes pour les tables de la base de données, chacune représentant une table et comprenant les relations avec les autres tables. Ces classes sont dérivées de Base, qui est une classe créée par automap_base. Les tables mappées comprennent `"user"`, `"session"`, `"module"`, `"ue"`, `"role"`, `"favorite"`, `"managed_by"`, `"lectured_by"`, et `"session_type"`.

La fonction connect_db récupère l'URL de connexion à la base de données à partir de la configuration Flask, crée un moteur de base de données en utilisant `create_engine`, puis utilise MetaData pour refléter les métadonnées de la base de données. Ensuite, il crée une instance Base en utilisant `automap_base` et définit des classes pour chaque table mappée.

Enfin, la fonction `connect_db` crée une session SQLAlchemy en utilisant `sessionmaker` et `flask_scoped_session`, qui fournit une session spécifique à la demande pour chaque requête, puis retourne la session SQLAlchemy.

auth.py :

La fonction `login_required` est un décorateur qui permet de protéger l'accès à certaines pages de l'application en obligeant l'utilisateur à être connecté. Cette fonction vérifie si l'utilisateur est connecté en vérifiant la présence d'un nom d'utilisateur dans la session. Si l'utilisateur n'est pas connecté, la fonction redirige l'utilisateur vers la page de connexion. Si l'utilisateur est connecté, la fonction appelle la fonction décorée.

La fonction `admin_required` est un autre décorateur qui protège l'accès à certaines pages en obligeant l'utilisateur à être connecté en tant qu'administrateur. Cette fonction vérifie si l'utilisateur est un administrateur en vérifiant la valeur de la propriété admin de l'objet `User` stocké dans `g.user`. Si l'utilisateur n'est pas un administrateur, la fonction renvoie une erreur 403 (interdiction). Si l'utilisateur est un administrateur, la fonction appelle la fonction décorée.

La fonction `load_logged_in_user` est un gestionnaire d'événements qui s'exécute avant chaque requête HTTP entrante. Cette fonction charge l'utilisateur connecté à partir de la base de données et stocke l'objet `User` dans `g.user`. Si aucun utilisateur n'est connecté, `g.user` est défini sur `None`. Si l'utilisateur n'est pas connecté et qu'il essaie d'accéder à une page autre que la page de connexion, la fonction redirige l'utilisateur vers la page de connexion.

La route `/register` affiche une page qui permet à un utilisateur de s'enregistrer s'il n'est pas encore enregistré dans la base de données. Si l'utilisateur est déjà enregistré et toutes les informations sont correctes, la fonction redirige l'utilisateur vers la page d'accueil. Si l'utilisateur est déjà enregistré mais certaines informations manquent, la page d'inscription est affichée pour que l'utilisateur puisse fournir ces informations manquantes.

La route `/login` affiche la page de connexion.

La route `/logout` déconnecte l'utilisateur en supprimant toutes les informations de session et en redirigeant l'utilisateur vers la page de connexion.

data.py :

Ce code permet de faire un processing des données reçues en entrée par l'extrait de Synapses sous format JSON et permet de remplir la base de données selon le schéma que nous avons choisi pour notre application.

Le script définit une route `/update` qui peut être utilisée pour télécharger un fichier JSON et mettre à jour la base de données en fonction de son contenu.

Les détails du format JSON attendu sont documentés dans la fonction `update()` - le fichier JSON doit contenir une liste d'objets, chaque objet représentant une session, qui contient des informations telles que l'ID de la session, l'UE, le module, les intervenants, les groupes, les salles, l'heure de début et de fin, etc.

Le code utilise pandas pour lire le fichier JSON et le transformer en plusieurs DataFrames. Il utilise ensuite SQLAlchemy pour insérer les données dans la base de données. Le code définit également une fonction `parse_users()` pour extraire les informations de nom, prénom et nom d'utilisateur à partir des champs d'intervenant du fichier JSON.

La fonction `read_json()` est utilisée pour lire le fichier JSON et renvoyer les données sous forme de tuple de DataFrames et de dictionnaires. Les DataFrames contiennent les informations sur les sessions, les utilisateurs et les entités liées (modules, UE, types de session). Les dictionnaires sont utilisés pour stocker les relations entre ces entités.

`home.py :`

Le code définit une vue (ou route) pour l'URL racine de l'application (`"/"`). Cette vue est décorée avec le décorateur "`@login_required`", qui indique que l'utilisateur doit être authentifié pour accéder à cette page.

La vue récupère ensuite une liste de sessions urgentes (i.e. sans tuteurs) et la liste des sessions à venir de l'utilisateur courant. Ces informations sont récupérées à partir de la base de données à l'aide de SQLAlchemy.

La vue calcule également le temps total que l'utilisateur a passé à enseigner, en additionnant la durée de toutes les sessions qu'il a enseignées. La durée est stockée dans un objet `datetime.timedelta`.

Enfin, la vue appelle la fonction "`render_template`" pour générer la page HTML à renvoyer au client. Les variables "`urgent_sessions`", "`user_next_sessions`" et "`total_duration`" sont transmises à cette fonction pour être utilisées dans le modèle HTML.

`profile.py :`

Ce code définit une route `/profile` qui permet à l'utilisateur de mettre à jour son profil. Il vérifie que l'utilisateur est connecté en utilisant la fonction de décoration `@login_required` importée depuis `tutorlink.auth`. La vue `profile()` peut être appelée soit en utilisant la méthode GET pour afficher le formulaire de mise à jour de profil, soit en utilisant la méthode POST pour valider la mise à jour du profil.

La vue récupère les informations de l'utilisateur connecté via l'objet `g.user` fourni par Flask et utilise la méthode `request.form` pour récupérer les informations du formulaire soumis par l'utilisateur.

La vue effectue ensuite une validation sur les champs du formulaire soumis et retourne une erreur en cas de problème de validation. Si tout est valide, la vue met à jour les informations du profil de l'utilisateur, sauvegarde les changements dans la base de données et redirige l'utilisateur vers la page de profil.

Le formulaire de mise à jour de profil inclut également une liste déroulante pour sélectionner un rôle. La vue récupère la liste des rôles disponibles depuis la base de données et les affiche dans la liste déroulante.

`session.py :`

Le code définit deux vues.

`session_list`, qui affiche toutes les sessions et permet à l'utilisateur de les filtrer par type, module, UE et dates. Les sessions sont paginées avec 12 sessions par page. Cette vue nécessite une authentification.

`session_register`, qui permet à l'utilisateur de s'inscrire à une session ou d'inscrire d'autres utilisateurs à une session si l'utilisateur est un administrateur ou gère le module de la session. Cette vue nécessite une authentification et une requête POST avec un ID de session.

Le code définit également une fonction auxiliaire, `pages_list`, qui renvoie une liste de pages à afficher dans la barre de pagination de la vue `session_list`.

`admin_panel.py :`

Ce blueprint sert pour la partie de l'application réservée aux administrateurs.

Le blueprint contient 3 routes : `"/panel"` et `"/grant_admin/<username>"`, et `"/revoke_admin/<username>"`.

"/panel" est une vue qui affiche le panneau d'administration de l'application. La vue récupère une liste de tous les utilisateurs de l'application à partir de la base de données et la renvoie sous forme de paramètre au modèle Jinja2.

"/grant_admin/<username>" est une vue qui permet à un administrateur de donner à un utilisateur spécifique des privilèges d'administrateur. La vue prend le nom d'utilisateur de l'utilisateur à qui les privilèges doivent être accordés en tant que paramètre et effectue la modification correspondante dans la base de données. Si l'utilisateur n'existe pas, un message d'erreur est renvoyé. Si l'utilisateur a déjà des privilèges d'administrateur, un message d'avertissement est renvoyé. Si l'opération réussit, un message de succès est renvoyé et l'utilisateur est redirigé vers "/panel".

"/revoke_admin/<username>" fait exactement la même chose que la route dessus mais pour enlever les droits d'administrateurs.

Le code utilise également des décorateurs tels que "login_required" et "admin_required" pour assurer la sécurité et limiter l'accès aux utilisateurs ayant les privilèges nécessaires. En effet, toutes ces routes nécessitent les droits d'administrateur. Enfin, nous utilisons "flash" pour afficher des messages à l'utilisateur après une action réussie ou un échec.